BME646 and ECE 60146 – Homework 4

Nadine Amin

Section 3.1

To create a subset of the COCO dataset, the COCO API is used, and an instance is initialized the downloaded annotation file. The COCO API is used to get the ID of each of the 5 desired categories, and the IDs of images belonging to each of those categories. In the COCO dataset, a certain image can have multiple labels (i.e. belong to multiple categories). However, for the sake of our subset dataset, only images that belong to only one of the 5 categories are used. Hence, for each of the 5 categories, the corresponding list of image IDs is cleaned to remove IDs that appear in any of the other lists (belonging to the other categories). See Figure 1.

```
# specify annotation file
annotation_file = [redacted]/ECE 60146 - Deep Learning/train2017/annotations/instances_train2017.json'

# COCO API - initialize with annotation file
my_coco = COCO(annotation_file)

loading annotations into memory...
Done (t=29.54s)
creating index...
index created!

# specify the 5 desired categories
my_categories = ['boat', 'cake', 'couch', 'dog', 'motorcycle']
# get IDs of the 5 desired categories
catIDs = []
for category in my_categories:
  catIDs.append(my_coco.getCatIds(catNms = category)[0])

# get IDs of images from each of the 5 desired categories
all_imgIDs = []
for i in range(len(catIDs)):
  all_imgIDs.append(my_coco.getImgIds(catIds = catIDs[i]))

# remove images belonging to multiple categories
my_imgIDs = {}
my_imgIDs[my_categories[0]] = list(set(all_imgIDs[0]) - set(all_imgIDs[1] + all_imgIDs[2] + all_imgIDs[3] + all_imgIDs[4]))
my_imgIDs[my_categories[1]] = list(set(all_imgIDs[1]) - set(all_imgIDs[0] + all_imgIDs[2] + all_imgIDs[3] + all_imgIDs[4]))
my_imgIDs[my_categories[2]] = list(set(all_imgIDs[2]) - set(all_imgIDs[0] + all_imgIDs[1] + all_imgIDs[3] + all_imgIDs[4]))
my_imgIDs[my_categories[3]] = list(set(all_imgIDs[3]) - set(all_imgIDs[0] + all_imgIDs[1] + all_imgIDs[2] + all_imgIDs[4]))
my_imgIDs[my_categories[4]] = list(set(all_imgIDs[4]) - set(all_imgIDs[0] + all_imgIDs[1] + all_imgIDs[2] + all_imgIDs[3]))
```

Figure 1

A dataset dictionary is initialized with empty lists for the training and validation splits. For each of our categories, 2000 indices are randomly generated without replacement (minimum possible value: 0, maximum possible value: the total number of images only belonging to this category minus 1). This is done using *numpy.random.Generator.choice*. The first 1600 indices are reserved for training instances and the last 400 for validation instances. (See Figure 2). Images are then processed for each split using the function *process_imgs* shown in Figure 3.

```
# initialize a dataset dictionary
my_dataset_dict = {}
my_dataset_dict['train'] = []
my_dataset_dict['val'] = []
# specify dataset directory
my_dataset_dir = [redacted]/ECE 60146 - Deep Learning/my_dataset/'

# for every category
for category in my_categories:

  # get image IDs for this category
  cat_imgIDs = my_imgIDs[category]

  # initialize a random generator to select a subset (2000) of the category images
  my_rand_gen = np.random.default_rng()
  # generate 2000 indices without replacement
  rnd_indices = my_rand_gen.choice(len(cat_imgIDs), 2000, replace = False)

  # let training IDs be the first 1600 randomly generated indices
  train_ids = rnd_indices[:1600]
  # let validation IDs be the last 400 randomly generated indices
  val_ids = rnd_indices[1600:]

  # process images for both splits (read, resize, and save images) + update dataset dictionary
  process_imgs(cat_imgIDs, train_ids, 'train')
  process_imgs(cat_imgIDs, val_ids, 'val')
```

Figure 2

The *splitIDs* input to the *process_imgs* function contains the randomly generated indices for images in the corresponding split. Therefore, for each dataset instance, the actual ID is extracted at the randomly generated index from the list of image IDs in that category. For each image, the image is loaded from the COCO URL, resized using *opencv*, named with its ID value, and saved to the dataset directory. The dataset dictionary is then updated such that a new element is added to the corresponding split. This element is a new dataset instance with *img* being the name of the image and *label* being the name of its category.

```
# a function that processes dataset images of a correspinding split
# the function loads images from COCO url, resizes them, saves them to dataset directory, and updates the dataset dictionary
def process_imgs(allIDs, splitIDs, split):

  # for every instance in the split
  for instance in splitIDs:

    # get instance ID
    ID = allIDs[instance]

    # get image at current ID using COCO API
    img_at_ID = my_coco.loadImgs(ID)[0]

    # read image from COCO url
    img_at_ID = io.imread(img_at_ID['coco_url'])

    # resize image to 64x64
    img_at_ID = cv2.resize(img_at_ID, (64, 64), interpolation = cv2.INTER_AREA)

    # save image to dataset directory
    img_name = str(ID) + '.jpg'
    cv2.imwrite(my_dataset_dir + split + '/' + img_name, img_at_ID)

    # create a dictionary for this dataset instance
    img = {}
    img['img'] = img_name
    img['label'] = category

    # add instance to the corresponding split of the dataset dictionary
    my_dataset_dict[split].append(img)
```

Figure 3

Three images from each category are plotted (see Figure 4) and shown in Figure 5.

```
# plot 3 training images from each category

# initialize a figure
fig = plt.figure(figsize=(4, 8))
sub_to_plot = 1

# for each category
for i in range(5):

  # generate random indices for each category (note: start and end points are because of the dataset order)
  rnd_idx = np.random.randint((i*1600), (i+1)*(1600), size = 3)

  # for each of the three images
  for j in range(3):

    # get image name from training split
    img_to_plot_name = my_dataset_dict['train'][rnd_idx[j]]['img']

    # read image from path
    img_to_plot = cv2.imread(my_dataset_dir + 'train/' + img_to_plot_name)

    # plot image
    fig.add_subplot(5, 3, sub_to_plot)
    plt.imshow(img_to_plot)
    plt.axis('off')
    sub_to_plot += 1
```

Figure 4

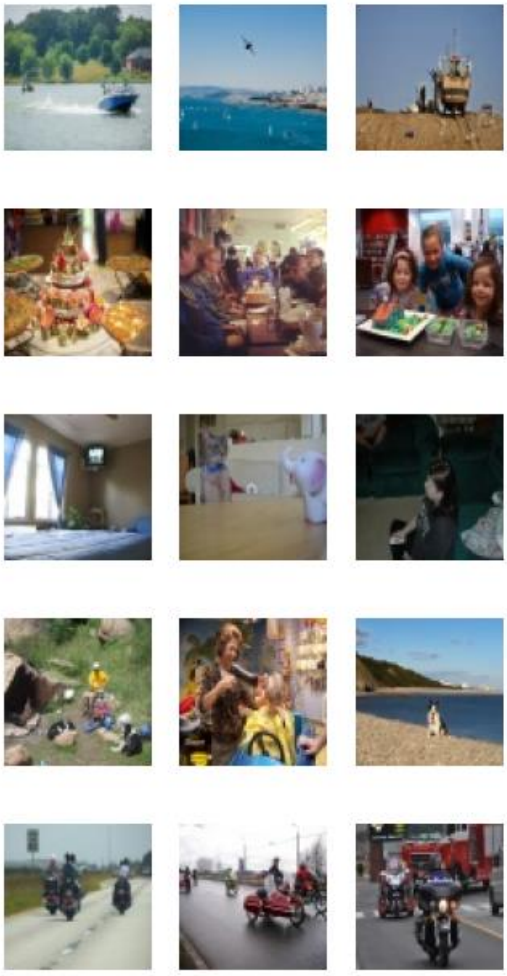

Figure 5

The dataset instances are then shuffled, a list of the category names is added to the dataset dictionary, and the dataset dictionary is then saved as a json file as shown in Figure 6.

```
# shuffle dataset instances
np.random.shuffle(my_dataset_dict['train'])
np.random.shuffle(my_dataset_dict['val'])

# add categories to dataset dictionary
my_dataset_dict['categories'] = ['boat', 'cake', 'couch', 'dog', 'motorcycle']

# specify name for dataset file
my_dataset_file = my_dataset_dir + 'my_dataset.txt'

# save dataset dictionary as a json file
with open(my_dataset_file, 'w') as json_file:
  json.dump(my_dataset_dict, json_file)
```

Figure 6

Section 3.2

Dataset Class

In order to load the subset of the COCO dataset created in Section 3.1, a dataset class is needed. I used the dataset class I had implemented for Homework 2 and adapted it as needed. Figure 7 shows the dataset class. The *__init__()* function takes as input (1) the split (being 'train' or 'val') to determine which images should be included in the dataset and (2) the root directory of the dataset. It accordingly updates the values of the *self.split* and *self.root* instance variables. In addition, it loads the dataset dictionary from the root directory and extracts category names. Lastly, it specifies the transformations that need to be made to training and validation images. For training purposes, images are horizontally flipped with a probability of 0.5. For both training and validation purposes, *tvt.Grayscale(num_output_channels = 3)* is used to ensure all images have 3 output channels. This is important because some of the images in the dataset are originally grayscale. Lastly, for both training and validation images, we get the tensor representation, as well as perform pixel value scaling and normalization. The *__len__()* function returns the number of images of the chosen split as indicated by the dataset dictionary. In the *__getitem__()* function, the name of the image at the desired index is read from the dataset dictionary. Next, the image is read using PIL and the appropriate transformations are carried out depending on the split. The image label is also read from the dataset dictionary and, using the category names, transformed into a new label with the category index. Lastly, the transformed image and the index label are returned.

```python
# dataset class
class MyDataset(torch.utils.data.Dataset):
  def __init__ (self, split, root):
    super().__init__()

    # initialize the split and the root path with None
    self.split = None
    self.root = None

    # if chosen split is training
    if split == 'train':
      # set root directory to be that of training
      self.root = root + 'train/'
      # chosen split
      self.split = 'train'
    if split == 'val':
      # root path for validation images
      self.root = root + 'val/'
      # chosen split
      self.split = 'val'

    # load dataset dictionary
    with open(root + 'my_dataset.txt') as json_file:
      self.dataset_dict = json.load(json_file)
    self.categories = self.dataset_dict['categories']

    # data augmentation transforms for training data
    self.train_trans = tvt.Compose([
        # randomly flip dataset images horizontally
        tvt.RandomHorizontalFlip(p=0.5),
        # transform number of channels into 3 (this is important because some of the instances are in grayscale)
        tvt.Grayscale(num_output_channels = 3),
        # get tensor representation of dataset images and perform pixel value scaling
        tvt.ToTensor(),
        # perform pixel normalization
        tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    # transforms for validation data (without augmentation)
    self.val_trans = tvt.Compose([
        # transform number of channels into 3 (this is important because some of the instances are in grayscale)
        tvt.Grayscale(num_output_channels = 3),
        # get tensor representation of dataset images and perform pixel value scaling
        tvt.ToTensor(),
        # perform pixel normalization
        tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

  def __len__(self):
    # return the total number of images
    return len(self.dataset_dict[self.split])

  def __getitem__(self, index):
    # get name of image at specified index from specified split
    img_name = self.dataset_dict[self.split][index]['img']

    # read image
    img = Image.open(self.root + img_name)

    # perform appropriate transforms on image (depending on the split)
    trans_img = None
    if (self.split == 'train'):
      trans_img = self.train_trans(img)
    if (self.split == 'val'):
      trans_img = self.val_trans(img)

    # get categorical image label
    categ_label = self.dataset_dict[self.split][index]['label']
    # set label to the corresponding class index
    label = self.categories.index(categ_label)

    # return augmented tensor and integer label
    return trans_img, label
```

Figure 7

Task 1

Since our dataset has 5 classes, the value of XX in Net 1 (and all other networks) is 5. To calculate XXXX for Net 1, we calculate the size of the output after each layer:

- Input Size : 3×64×64
- 1st Convolution:
  - Input Channels: 3
  - Output Channels: 16
  - Kernel Size: 3
  - Stride: 1 (default)
  - Padding: 0 (default)
  - Output $= \frac{N+(2*P)-K}{S} + 1 = \frac{64+(2*0)-3}{1} + 1 = 62$
  - Output Size: 16×62×62
- 1st Max Pooling (2D):
  - Output Size: $16 \times \frac{62}{2} \times \frac{62}{2}$ =16×31×31
- 2nd Convolution:
  - Input Channels: 16
  - Output Channels: 32
  - Kernel Size: 3
  - Stride: 1 (default)
  - Padding: 0 (default)
  - Output $= \frac{N+(2*P)-K}{S} + 1 = \frac{31+(2*0)-3}{1} + 1 = 29$
  - Output Size: 32×29×29
- 2nd Max Pooling (2D):
  - Output Size: $32 \times \left\lfloor \frac{29}{2} \right\rfloor \times \left\lfloor \frac{29}{2} \right\rfloor = 32 \times 14 \times 14$
- Linear Layer:
  - Input Size (after flattening) = 32×14×14 = 6272

Therefore, XXXX is 6272.

Figure 8 shows the class for Task 1, named HW4Net1. It is copied from the homework guidelines, and the values for XX and XXXX are updated.

```python
# network class for task 1
# copied from homework guidelines
class HW4Net1(nn.Module):
  def __init__(self):
    super(HW4Net1, self).__init__()
    # 1st convolution layer
    self.conv1 = nn.Conv2d(3, 16, 3)
    # pooling layer
    self.pool = nn.MaxPool2d(2, 2)
    # 2nd convolution layer
    self.conv2 = nn.Conv2d(16, 32, 3)
    # 1st fully connected layer
    self.fc1 = nn.Linear(6272, 64)
    # 2nd fully connected layer
    self.fc2 = nn.Linear(64, 5)

  def forward(self, x):
    # 1st convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv1(x)))
    # 2nd convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(x.shape[0], -1)
    # 1st fully connected layer + ReLU activation
    x = F.relu(self.fc1(x))
    # 2nd fully connected layer
    x = self.fc2(x)
    return x
```

Figure 8

Task 2

Again, the value of XX in Net 2 (like all other networks) is 5. To calculate XXXX for Net 2, we calculate the size of the output after each layer:

- Input Size : 3×64×64
- 1st Convolution:
  - Input Channels: 3
  - Output Channels: 16
  - Kernel Size: 3
  - Stride: 1 (default)
  - Padding: 1
  - Output $= \frac{N+(2*P)-K}{S} + 1 = \frac{64+(2*1)-3}{1} + 1 = 64$
  - Output Size: 16×64×64
- 1st Max Pooling (2D):
  - Output Size: $16\times\frac{64}{2}\times\frac{64}{2}$ =16×32×32
- 2nd Convolution:
  - Input Channels: 16
  - Output Channels: 32
  - Kernel Size: 3
  - Stride: 1 (default)
  - Padding: 1

- Output $= \frac{N+(2*P)-K}{S} + 1 = \frac{32+(2*1)-3}{1} + 1 = 32$
- Output Size: 32×32×32

- 2nd Max Pooling (2D):
  - Output Size: $32 \times \frac{32}{2} \times \frac{32}{2} = 32 \times 16 \times 16$
- Linear Layer:
  - Input Size (after flattening) = 32×16×16 = 8192

Therefore, XXXX is 8192.

Figure 9 shows the class for Task 2, named HW4Net2. It is an updated version of HW4Net1 such that each convolution layer includes a padding of 1. The value of XXXX is also updated.

```
# network class for task 2
# updated from homework guidelines
class HW4Net2(nn.Module):
  def __init__(self):
    super(HW4Net2, self).__init__()
    # 1st convolution layer
    self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)
    # pooling layer
    self.pool = nn.MaxPool2d(2, 2)
    # 2nd convolution layer
    self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
    # 1st fully connected layer
    self.fc1 = nn.Linear(8192, 64)
    # 2nd fully connected layer
    self.fc2 = nn.Linear(64, 5)

  def forward(self, x):
    # 1st convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv1(x)))
    # 2nd convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(x.shape[0], -1)
    # 1st fully connected layer + ReLU activation
    x = F.relu(self.fc1(x))
    # 2nd fully connected layer
    x = self.fc2(x)
    return x
```

Figure 9

Task 3

Again, the value of XX in Net 3 (like all other networks) is 5. To calculate XXXX for Net 3, we calculate the size of the output after each layer. The first [Convolution + Pooling + Convolution + Pooling] are exactly the same as in Task 2. Hence, their output is of size 32×16×16. The extra 10 convolution layers added are each of 32 input channels, 32 output channels, a kernel size of 3, and a padding of 1. Following the equation: Output $= \frac{N+(2*P)-K}{S}$, we see that $\frac{N+(2*1)-3}{1} + 1 = \frac{N-1}{1} + 1 = N$. This means that the output size will be equal to the input size. Therefore, the output after the chained convolution layers will be 32×16×16. Therefore, the input size to the linear layer, i.e. XXXX, would also be 32×16×16 = 8192.

Figure 10 shows the class for Task 3, named HW4Net3. It is an updated version of HW4Net2 such that an instance variable called *self.conv_extra* stores a list of the 10 convolution layers using *nn.ModuleList()*. In the *forward()* function, the output of the first two convolution layers and poolings is passed through each of the 10 extra convolution layers, each followed by the nonlinear ReLU activation function. The final output is then passed through the linear layers.

```
# network class for task 3
# updated from homework guidelines
class HW4Net3(nn.Module):
  def __init__(self):
    super(HW4Net3, self).__init__()
    # 1st convolution layer
    self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)
    # pooling layer
    self.pool = nn.MaxPool2d(2, 2)
    # 2nd convolution layer
    self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
    # 10 extra convolution layers
    self.conv_extra = nn.ModuleList()
    for i in range(10):
      self.conv_extra.append(nn.Conv2d(32, 32, 3, padding = 1))
    # 1st fully connected layer
    self.fc1 = nn.Linear(8192, 64)
    # 2nd fully connected layer
    self.fc2 = nn.Linear(64, 5)

  def forward(self, x):
    # 1st convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv1(x)))
    # 2nd convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv2(x)))
    # 10 extra convolution layers + ReLU activation
    for i in range(10):
      x = F.relu(self.conv_extra[i](x))
    x = x.view(x.shape[0], -1)
    # 1st fully connected layer + ReLU activation
    x = F.relu(self.fc1(x))
    # 2nd fully connected layer
    x = self.fc2(x)
    return x
```

Figure 10

Training Networks

First, as shown in Figure 11, an instance of each of the network classes is created. The number of learnable parameters for each network is calculated using the function *get_num_params()* defined as shown in Figure 12. The syntax for calculating the number of parameters is inspired from DLStudio (https://engineering.purdue.edu/kak/distDLS/), where the *numel()* function is used on each of the network parameters to count the number of parameters that are learnable (requiring gradient). As shown in Figure 11, the numbers of parameters are 406885, 529765, and 622245 for Net1, Net2, and Net3 respectively (also see Table 1).

```
# instantiate the three networks
Net1 = HW4Net1()
Net2 = HW4Net2()
Net3 = HW4Net3()

print("Number of Parameters:\nNet1: " + str(get_num_params(Net1)) + "\nNet2: " + str(get_num_params(Net2)) + "\nNet3: " + str(get_num_params(Net3)))
```

```
Number of Parameters:
Net1: 406885
Net2: 529765
Net3: 622245
```

Figure 11

```
# a function that returns the total number of learnable parameters of a network
# reference: DLStudio (https://engineering.purdue.edu/kak/distDLS/)
def get_num_params(network):
  # sum the number of parameters that are learnable (requiring gradient)
  return sum(param.numel() for param in network.parameters() if param.requires_grad)
```

Figure 12

Next, as shown in Figure 13, each of the networks is trained and the calculated training losses are plotted on the same figure (see Figure 15). The *train_network()* function is defined as shown in Figure 14. The code is adapted from the example shown in the homework guidelines. The dataset directory is first specified, the device set to *cuda:0*, and the network moved to the device. Next, an instance of the dataset class is created with the split set to *'train'*. An instance of *torch.utils.data.DataLoader* is also created to wrap the dataset instance and set the batch size to 4, allow shuffling of dataset instances, as well as set the number of workers to 2. Next, the Cross Entropy loss is chosen, the optimizer is chosen to be Adam with a learning rate of 1e-3 and betas of default values (0.9 and 0.99). The number of epochs is chosen to be 10, and an empty list is initialized to later store the training loss every 100 iterations. In every epoch, and for each batch in the dataset, images and labels are read and moved to the device. The gradients of learnable parameters are reset to 0 and the inputs are passed through the network. The loss is then calculated, and back propagation is performed. Lastly, the optimizer step is updated, and the calculated loss added to the running loss. Every 100 iterations, the running loss is averaged, printed, and stored in the list of training losses to be returned by the function. The running loss is then reset to 0. At the very end, the final list of training losses every 100 iterations is returned.

```
# train the three networks
net1_training_loss = train_network(Net1)
net2_training_loss = train_network(Net2)
net3_training_loss = train_network(Net3)

# plot all training losses vs iterations
plt.figure()
plt.plot(net1_training_loss, label="Net1")
plt.plot(net2_training_loss, label="Net2")
plt.plot(net3_training_loss, label="Net3")
# set legend, figure title, and axes labels
plt.legend(loc="upper right")
plt.xlabel("Iterations")
plt.ylabel("Training Loss")
plt.title("Training Loss vs Iterations (10 Epochs)")
# show figure
plt.show()
```

Figure 13

```python
# a function that trains a network
# updated from homework guidelines
def train_network(network):

  # specify dataset root directory
  data_dir = [redacted]/ECE 60146 - Deep Learning/my_dataset/'
  # specify device
  device = torch.device("cuda:0")
  # move network to device
  network = network.to(device)

  # instantiate the dataset for training purposes
  my_dataset = MyDataset(split = 'train', root = data_dir)
  # wrap the dataset instance within torch.utils.data.DataLoader
  my_dataloader = torch.utils.data.DataLoader(my_dataset, batch_size = 4, shuffle = True, num_workers = 2)

  # specify criterion, optimizer, and number of epochs
  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(network.parameters(), lr = 1e-3, betas = (0.9, 0.99))
  epochs = 10

  # make an empty list to store training loss
  training_loss = []

  # for every epoch
  for epoch in range(epochs):

    # initialize running loss with 0.0
    running_loss = 0.0

    # for every batch in the dataset
    for i, data in enumerate(my_dataloader):
      # get batch images and labels
      inputs, labels = data
      # move images and labels to device
      inputs = inputs.to(device)
      labels = labels.to(device)

      # set gradients of learnable parameters to zero
      optimizer.zero_grad()
      # pass inputs into the network
      outputs = network(inputs)
      # calculate loss
      loss = criterion(outputs, labels)
      # perform back propagation
      loss.backward()
      # update optimizer step
      optimizer.step()
      # update running loss
      running_loss += loss.item()

      # save loss every 100 iterations
      if ((i+1) % 100 == 0):
        # get average of running loss
        running_loss /= 100
        # add loss to list of training losses
        training_loss.append(running_loss)
        # print current epoch, batch, and averaged loss
        print ("[epoch: %d, batch: %5d] loss: %.3f" \
              % (epoch + 1, i + 1, running_loss))
        # reset running loss
        running_loss = 0.0

  # return training loss
  return training_loss
```

Figure 14

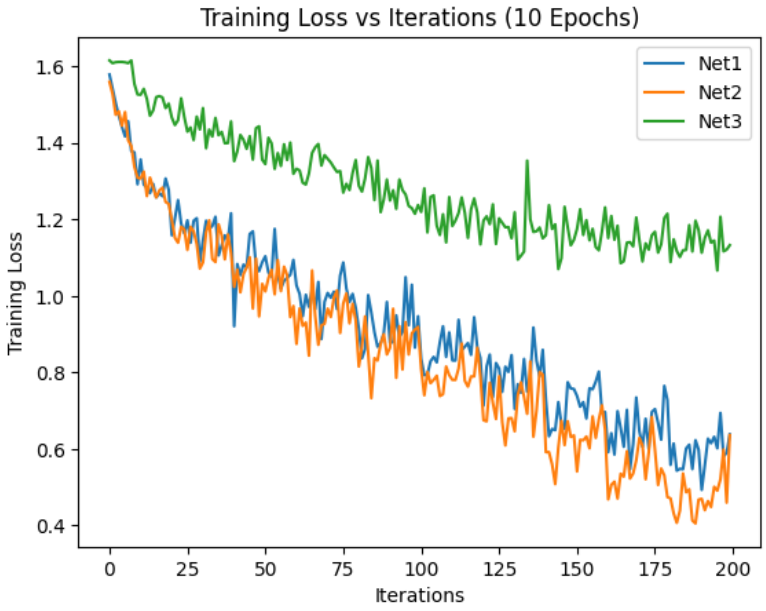

Figure 15 – (iterations in hundreds)

Figure 15 shows the resulting plot of the training losses per iterations (in hundreds) for each of the three networks. As can be seen, the training loss curves of Net1 and Net2 are very close. The training loss of Net3, on the other hand, is much larger than that of the other two networks. More on these differences is explained below (see Comparison & Questions section).

Testing

After training the networks, they are tested on the validation dataset. The function *test_network()* is implemented as shown in Figure 17 and is inspired from DLStudio (https://engineering.purdue.edu/kak/distDLS/). The dataset directory is first specified, the device set to *cuda:0*, and the network set to evaluation mode and moved to the device. Next, an instance of the dataset class is created with the split set to *'val*. An instance of *torch.utils.data.DataLoader* is also created to wrap the dataset instance and set the batch size to 4, allow shuffling of dataset instances, as well as set the number of workers to 2. Empty lists are initialized to later store the true labels and corresponding labels predicted by the network. Next, *with torch.no_grad()* is used to indicate that gradient calculation is disabled. Then, for every batch in the validation dataset, the inputs and true labels are read and moved to the device. They are then passed through the network and the output is calculated. The output contains a value for each of our 5 classes; hence, the predicted class is extracted to be the one assigned a greater value. Lastly, the lists of true and predicted labels are extended to include those of the current batch. Both lists (true and predicted labels) are finally returned by the function.

```
# test each of the three networks
t_net1, p_net1 = test_network(Net1)
t_net2, p_net2 = test_network(Net2)
t_net3, p_net3 = test_network(Net3)
```

Figure 16

```python
# a function that tests a network, inspired from DLStudio (https://engineering.purdue.edu/kak/distDLS/)
def test_network(network):
  # specify dataset root directory
  data_dir = [redacted]/ECE 60146 - Deep Learning/my_dataset/'
  # specify device
  device = torch.device("cuda:0")
  # set model in evaluation mode
  network = network.eval()
  # move network to device
  network = network.to(device)

  # instantiate the dataset for validation purposes
  my_dataset = MyDataset(split = 'val', root = data_dir)
  # wrap the dataset instance within torch.utils.data.DataLoader
  my_dataloader = torch.utils.data.DataLoader(my_dataset, batch_size = 4, shuffle = True, num_workers = 2)

  # create empty lists to store true and predicted labels
  true_labels = []
  predicted_labels = []

  # indicate disabling gradient calculation
  with torch.no_grad():
    # for every batch in the dataset
    for i, data in enumerate(my_dataloader):

      # get batch images and labels
      inputs, t_labels = data
      # move batch images and labels to device
      inputs = inputs.to(device)
      t_labels = t_labels.to(device)

      # pass inputs into the network
      outputs = network(inputs)
      # get label with maximum output value
      max_value, p_labels = torch.max(outputs.data, 1)

      # add true and predicted labels to corresponding lists
      true_labels.extend(t_labels.tolist())
      predicted_labels.extend(p_labels.tolist())

  # return lists of true and predicted labels
  return true_labels, predicted_labels
```

Figure 17

After obtaining the lists of true and predicted labels for each of the networks, the validation accuracy is calculated using the *accuracy_score function* from *sklearn.metrics* (imported here as *accuracy*). As shown in Figure 18, the validation accuracies are 54.35%, 54.30%, and 51.05% for Net1, Net2, and Net3 respectively (also see Table 1). We can see that Net1 and Net2 achieved almost the same accuracy, while Net3 achieved a slightly less score. More on these differences is explained below (see Comparison & Questions section). Lastly, as shown in Figure 19, the confusion matrix for each of the networks is calculated and displayed using the *ConfusionMatrixDisplay* function from *sklearn.metrics* (imported here as *cfd*). Figures 20, 21, and 22 show the confusion matrices for each of the networks.

```
# calculate validation accuracy of each of the three networks
val_acc_1 = accuracy(t_net1, p_net1)
val_acc_2 = accuracy(t_net2, p_net2)
val_acc_3 = accuracy(t_net3, p_net3)
print("Validation Accuracy:\nNet1: " + str(val_acc_1) + "\nNet2: " + str(val_acc_2) + "\nNet3: " + str(val_acc_3))
```

```
Validation Accuracy:
Net1: 0.5435
Net2: 0.543
Net3: 0.5105
```

Figure 18

```
# calculate confusion matrices of each of the three networks
my_categories = ['boat', 'cake', 'couch', 'dog', 'motorcycle']
conf_mat_1 = cfd.from_predictions(t_net1, p_net1, display_labels = np.array(my_categories), cmap='GnBu')
conf_mat_2 = cfd.from_predictions(t_net2, p_net2, display_labels = np.array(my_categories), cmap='GnBu')
conf_mat_3 = cfd.from_predictions(t_net3, p_net3, display_labels = np.array(my_categories), cmap='GnBu')
```

Figure 19

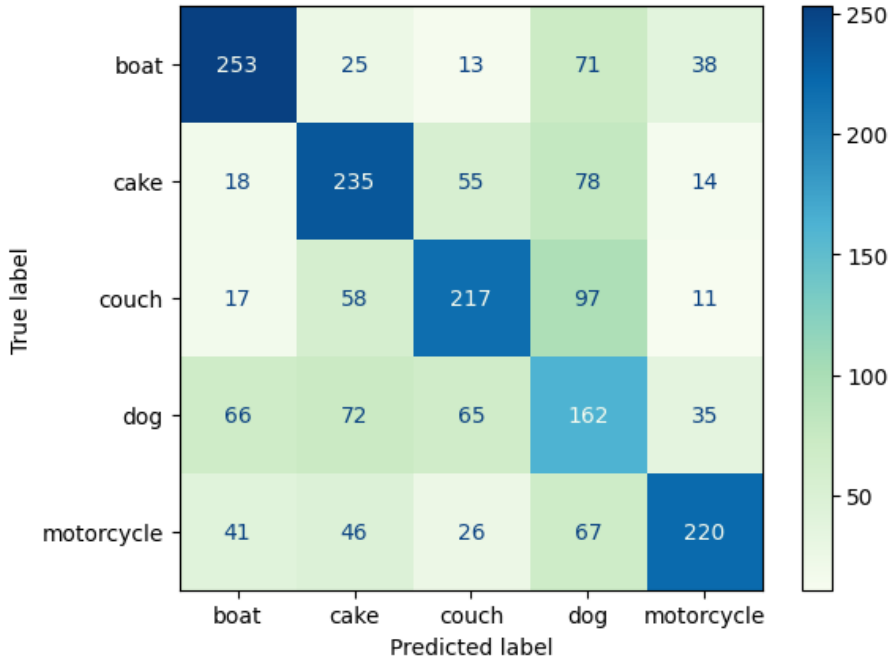

Figure 20 – Net1

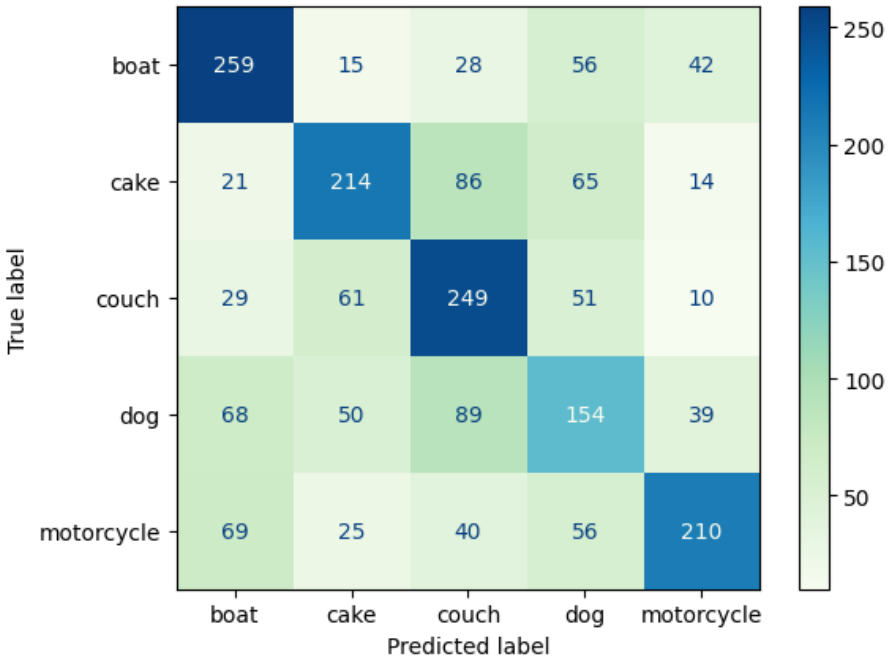

Figure 21 – Net2

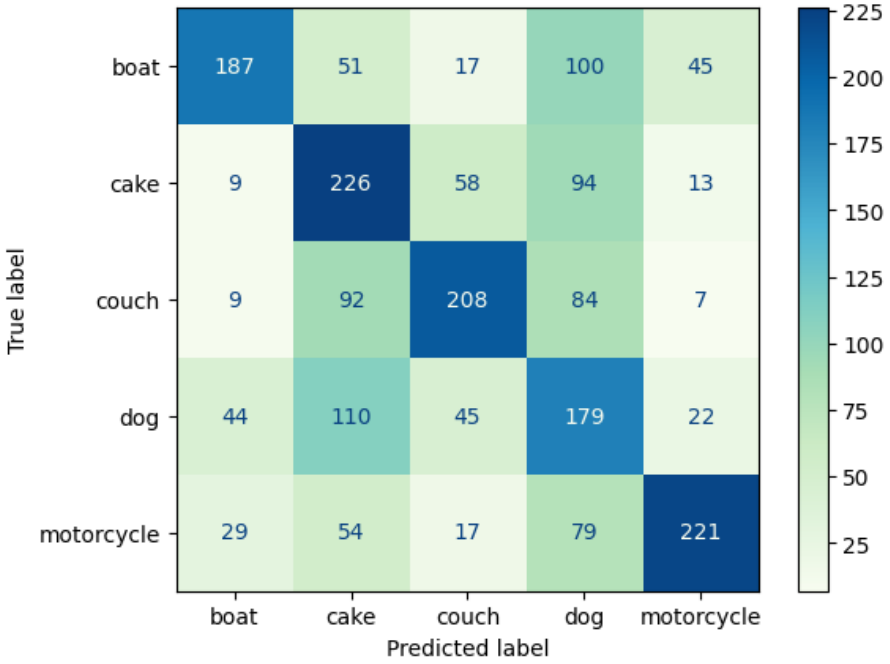

Figure 22 – Net3

Comparison & Questions

Table 1 summarizes the number of parameters and the classification accuracy for each of the three networks. Lastly, the questions in Homework 4 are answered below.

	# Parameters	Validation Accuracy
Net 1	406885	54.35%
Net 2	529765	54.30%
Net 3	622245	51.05%

Table 1

- **Does adding padding to the convolutional layers make a difference in classification performance?**
  - ➔ To compare the effect of adding padding to the convolutional layers, we compare the performance of Net1 and Net2. As can be seen from Figure 15, the training loss curves of Net1 and Net2 are very close. With the curve for Net2 very slightly below that of Net1, perhaps it converged just a little bit faster than Net1. However, the difference is probably insignificant. Similarly, from Figure 18 and Table 1, we can see that Net1 and Net2 achieved almost the same validation accuracy. Generally speaking, when padding is not added (as is the case with Net1), the size of the image shrinks as it goes through the network (see image size calculations in Task 1 section). On the other hand, adding a padding of 1 (as is the case with Net2) preserves the image size as it goes through the network (see image size calculations in Task 2 section). This preservation of size can result in enhanced performance. In our case, the image size did not shrink a lot in Net1 since the network contains only 2 convolution layers. Hence, the final performance of Net1 and Net2 was very close.
- **As you may have known, naively chaining a large number of layers can result in difficulties in training. This phenomenon is often referred to as vanishing gradient. Do you observe something like that in Net3?**
  - ➔ In Net3, 10 extra convolution layers are naively stacked after the first 2 convolution + pooling layers. As shown in Figure 15, the training loss of Net3 is much larger than that of the other two networks. The convergence is comparably slow, and the final loss is more than double the final loss of Net1 and Net2. Also, from Figure 18 and Table 1, we can see that Net3 had the lowest classification accuracy despite being the larger network. The reason for this is the vanishing gradient problem associated with the naïve increase in number of layers. During backpropagation, the propagated gradient becomes smaller and smaller until it somewhat vanishes. Hence, the updates in the parameter values are very small and the network becomes hard to train.
- **Compare the classification results by all three networks, which CNN do you think is the best performer?**
  - ➔ As discussed in detail in answers to questions 1 and 2, Net1 and Net2 performed quite similarly while Net3 had a lower performance in comparison. Therefore, Net1 and Net2 are preferred over Net3 (which suffers from the vanishing gradient problem). Because of the preservation of spatial dimensions through pooling, Net2 can probably be preferred over Net1.
- **By observing your confusion matrices, which class or classes do you think are more difficult to correctly differentiate and why?**

➔ In all three networks, the class that was hardest to differentiate is the *dog* class. As highlighted in red boxes in Figures 20 – 22, only 162, 154 and 179 out of 400 images were correctly classified as *dog* by the three networks respectively. Perhaps the reason can be that dogs can appear in images of very different contexts. In addition, most of the time, they only take up a small portion of the image due to their small size.

- **What is one thing that you propose to make the classification performance better?**
  ➔ Despite the training loss of Net1 and Net2 being much lower than that of Net3 (see Figure 15), their validation accuracies are not that far apart (see Table 1). This might indicate that Net1 and Net2 had actually slightly overfitted the training dataset. Perhaps it is worth experimenting whether using fewer training epochs for Net1 and Net2 would result in a better classification accuracy on the validation split. Regularization techniques to avoid overfitting, such as adding dropout layers, can also be experimented with. Lastly, by visually inspecting the 64×64 images in the dataset, we can see that such reduction in size makes them hard to categorize even for us humans. Hence, not reducing the image size that much preserves more image features that can be helpful to the networks in classification.

Full Source Code:

Libraries:

```
# import libraries
from pycocotools.coco import COCO
import numpy as np
import skimage.io as io
import matplotlib.pyplot as plt
import cv2
import json
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as tvt
import os
from PIL import Image
from sklearn.metrics import accuracy_score as accuracy, ConfusionMatrixDisplay as
cfd
```

Section 3.1

```
# specify annotation file
annotation_file = 'ECE 60146 - Deep
Learning/train2017/annotations/instances_train2017.json'

# COCO API - initialize with annotation file
my_coco = COCO(annotation_file)
```

```
# specify the 5 desired categories
my_categories = ['boat', 'cake', 'couch', 'dog', 'motorcycle']
# get IDs of the 5 desired categories
catIDs = []
for category in my_categories:
  catIDs.append(my_coco.getCatIds(catNms = category)[0])

# get IDs of images from each of the 5 desired categories
all_imgIDs = []
for i in range(len(catIDs)):
  all_imgIDs.append(my_coco.getImgIds(catIds = catIDs[i]))

# remove images belonging to multiple categories
my_imgIDs = {}
my_imgIDs[my_categories[0]] = list(set(all_imgIDs[0]) - set(all_imgIDs[1] +
all_imgIDs[2] + all_imgIDs[3] + all_imgIDs[4]))
my_imgIDs[my_categories[1]] = list(set(all_imgIDs[1]) - set(all_imgIDs[0] +
all_imgIDs[2] + all_imgIDs[3] + all_imgIDs[4]))
my_imgIDs[my_categories[2]] = list(set(all_imgIDs[2]) - set(all_imgIDs[0] +
all_imgIDs[1] + all_imgIDs[3] + all_imgIDs[4]))
my_imgIDs[my_categories[3]] = list(set(all_imgIDs[3]) - set(all_imgIDs[0] +
all_imgIDs[1] + all_imgIDs[2] + all_imgIDs[4]))
my_imgIDs[my_categories[4]] = list(set(all_imgIDs[4]) - set(all_imgIDs[0] +
all_imgIDs[1] + all_imgIDs[2] + all_imgIDs[3]))

# a function that processes dataset images of a corresponding split
# the function loads images from COCO url, resizes them, saves them to dataset
directory, and updates the dataset dictionary
def process_imgs(allIDs, splitIDs, split):

  # for every instance in the split
  for instance in splitIDs:

    # get instance ID
    ID = allIDs[instance]

    # get image at current ID using COCO API
    img_at_ID = my_coco.loadImgs(ID)[0]

    # read image from COCO url
    img_at_ID = io.imread(img_at_ID['coco_url'])

    # resize image to 64x64
    img_at_ID = cv2.resize(img_at_ID, (64, 64), interpolation = cv2.INTER_AREA)
```

```
    # save image to dataset directory
    img_name = str(ID) + '.jpg'
    cv2.imwrite(my_dataset_dir + split + '/' + img_name, img_at_ID)

    # create a dictionary for this dataset instance with image name and one hot
encoding for label
    img = {}
    img['img'] = img_name
    img['label'] = np.zeros(len(my_categories))
    img['label'][my_categories.index(category)] = 1

    # add instance to the corresponding split of the dataset dictionary
    my_dataset_dict[split].append(img)

# initialize a dataset dictionary
my_dataset_dict = {}
my_dataset_dict['train'] = []
my_dataset_dict['val'] = []
# specify dataset directory
my_dataset_dir = r'ECE 60146 - Deep Learning/my_dataset/'

# for every category
for category in my_categories:

  # get image IDs for this category
  cat_imgIDs = my_imgIDs[category]

  # initialize a random generator to select a subset (2000) of the category
images
  my_rand_gen = np.random.default_rng()
  # generate 2000 indices without replacement
  rnd_indices = my_rand_gen.choice(len(cat_imgIDs), 2000, replace = False)

  # let training IDs be the first 1600 randomly generated indices
  train_ids = rnd_indices[:1600]
  # let validation IDs be the last 400 randomly generated indices
  val_ids = rnd_indices[1600:]

  # process images for both splits (read, resize, and save images) + update
dataset dictionary
  process_imgs(cat_imgIDs, train_ids, 'train')
  process_imgs(cat_imgIDs, val_ids, 'val')

# plot 3 training images from each category
```

```
# initialize a figure
fig = plt.figure(figsize=(4, 8))
sub_to_plot = 1

# for each category
for i in range(5):

  # generate random indices for each category (note: start and end points are
because of the dataset order)
  rnd_idx = np.random.randint((i*1600), (i+1)*(1600), size = 3)

  # for each of the three images
  for j in range(3):

    # get image name from training split
    img_to_plot_name = my_dataset_dict['train'][rnd_idx[j]]['img']

    # read image from path
    img_to_plot = cv2.imread(my_dataset_dir + 'train/' + img_to_plot_name)

    # plot image
    fig.add_subplot(5, 3, sub_to_plot)
    plt.imshow(img_to_plot)
    plt.axis('off')
    sub_to_plot += 1

# shuffle dataset instances
np.random.shuffle(my_dataset_dict['train'])
np.random.shuffle(my_dataset_dict['val'])

# add categories to dataset dictionary
my_dataset_dict['categories'] = ['boat', 'cake', 'couch', 'dog', 'motorcycle']

# specify name for dataset file
my_dataset_file = my_dataset_dir + 'my_dataset.txt'

# save dataset dictionary as a json file
with open(my_dataset_file, 'w') as json_file:
  json.dump(my_dataset_dict, json_file)
```

Section 3.2

Dataset Class

```
# dataset class
class MyDataset(torch.utils.data.Dataset):
  def __init__ (self, split, root):
    super().__init__()

    # initialize the split and the root path with None
    self.split = None
    self.root = None

    # if chosen split is training
    if split == 'train':
      # set root directory to be that of training
      self.root = root + 'train/'
      # chosen split
      self.split = 'train'
    if split == 'val':
      # root path for validation images
      self.root = root + 'val/'
      # chosen split
      self.split = 'val'

    # load dataset dictionary
    with open(root + 'my_dataset.txt') as json_file:
      self.dataset_dict = json.load(json_file)
    self.categories = self.dataset_dict['categories']

    # data augmentation transforms for training data
    self.train_trans = tvt.Compose([
        # randomly flip dataset images horizontally
        tvt.RandomHorizontalFlip(p=0.5),
        # transform number of channels into 3 (this is important because some of
the instances are in grayscale)
        tvt.Grayscale(num_output_channels = 3),
        # get tensor representation of dataset images and perform pixel value
scaling
        tvt.ToTensor(),
        # perform pixel normalization
        tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    # transforms for validation data (without augmentation)
    self.val_trans = tvt.Compose([
        # transform number of channels into 3 (this is important because some of
the instances are in grayscale)
        tvt.Grayscale(num_output_channels = 3),
```

```
        # get tensor representation of dataset images and perform pixel value
scaling
        tvt.ToTensor(),
        # perform pixel normalization
        tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

  def __len__(self):
    # return the total number of images
    return len(self.dataset_dict[self.split])

  def __getitem__(self, index):
    # get name of image at specified index from specified split
    img_name = self.dataset_dict[self.split][index]['img']

    # read image
    img = Image.open(self.root + img_name)

    # perform appropriate transforms on image (depending on the split)
    trans_img = None
    if (self.split == 'train'):
      trans_img = self.train_trans(img)
    if (self.split == 'val'):
      trans_img = self.val_trans(img)

    # get categorical image label
    categ_label = self.dataset_dict[self.split][index]['label']
    # set label to the corresponding class index
    label = self.categories.index(categ_label)

    # return augmented tensor and integer label
    return trans_img, label
```

Helper Functions

```
# a function that returns the total number of learnable parameters of a network
# reference: DLStudio (https://engineering.purdue.edu/kak/distDLS/)
def get_num_params(network):
  # sum the number of parameters that are learnable (requiring gradient)
  return sum(param.numel() for param in network.parameters() if
param.requires_grad)

# Commented out IPython magic to ensure Python compatibility.
# a function that trains a network
# updated from homework guidelines
def train_network(network):
```

```
  # specify dataset root directory
  data_dir = 'ECE 60146 - Deep Learning/my_dataset/'
  # specify device
  device = torch.device("cuda:0")
  # move network to device
  network = network.to(device)

  # instantiate the dataset for training purposes
  my_dataset = MyDataset(split = 'train', root = data_dir)
  # wrap the dataset instance within torch.utils.data.DataLoader
  my_dataloader = torch.utils.data.DataLoader(my_dataset, batch_size = 4, shuffle
= True, num_workers = 2)

  # specify criterion, optimizer, and number of epochs
  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(network.parameters(), lr = 1e-3, betas = (0.9,
0.99))
  epochs = 10

  # make an empty list to store training loss
  training_loss = []

  # for every epoch
  for epoch in range(epochs):

    # initialize running loss with 0.0
    running_loss = 0.0

    # for every batch in the dataset
    for i, data in enumerate(my_dataloader):
      # get batch images and labels
      inputs, labels = data
      # move images and labels to device
      inputs = inputs.to(device)
      labels = labels.to(device)

      # set gradients of learnable parameters to zero
      optimizer.zero_grad()
      # pass inputs into the network
      outputs = network(inputs)
      # calculate loss
      loss = criterion(outputs, labels)
      # perform back propagation
      loss.backward()
```

```
      # update optimizer step
      optimizer.step()
      # update running loss
      running_loss += loss.item()

      # save loss every 100 iterations
      if ((i+1) % 100 == 0):
        # get average of running loss
        running_loss /= 100
        # add loss to list of training losses
        training_loss.append(running_loss)
        # print current epoch, batch, and averaged loss
        print ("[epoch: %d, batch: %5d] loss: %.3f" \
#               % (epoch + 1, i + 1, running_loss))
        # reset running loss
        running_loss = 0.0

  # return training loss
  return training_loss

# a function that tests a network, inspired from DLStudio
(https://engineering.purdue.edu/kak/distDLS/)
def test_network(network):
  # specify dataset root directory
  data_dir = 'ECE 60146 - Deep Learning/my_dataset/'
  # specify device
  device = torch.device("cuda:0")
  # set model in evaluation mode
  network = network.eval()
  # move network to device
  network = network.to(device)

  # instantiate the dataset for validation purposes
  my_dataset = MyDataset(split = 'val', root = data_dir)
  # wrap the dataset instance within torch.utils.data.DataLoader
  my_dataloader = torch.utils.data.DataLoader(my_dataset, batch_size = 4, shuffle
= True, num_workers = 2)

  # create empty lists to store true and predicted labels
  true_labels = []
  predicted_labels = []

  # indicate disabling gradient calculation
  with torch.no_grad():
    # for every batch in the dataset
```

```
    for i, data in enumerate(my_dataloader):

      # get batch images and labels
      inputs, t_labels = data
      # move batch images and labels to device
      inputs = inputs.to(device)
      t_labels = t_labels.to(device)

      # pass inputs into the network
      outputs = network(inputs)
      # get label with maximum output value
      max_value, p_labels = torch.max(outputs.data, 1)

      # add true and predicted labels to corresponding lists
      true_labels.extend(t_labels.tolist())
      predicted_labels.extend(p_labels.tolist())

  # return lists of true and predicted labels
  return true_labels, predicted_labels
```

Task 1 Class

```
# network class for task 1
# copied from homework guidelines
class HW4Net1(nn.Module):
  def __init__(self):
    super(HW4Net1, self).__init__()
    # 1st convolution layer
    self.conv1 = nn.Conv2d(3, 16, 3)
    # pooling layer
    self.pool = nn.MaxPool2d(2, 2)
    # 2nd convolution layer
    self.conv2 = nn.Conv2d(16, 32, 3)
    # 1st fully connected layer
    self.fc1 = nn.Linear(6272, 64)
    # 2nd fully connected layer
    self.fc2 = nn.Linear(64, 5)

  def forward(self, x):
    # 1st convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv1(x)))
    # 2nd convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(x.shape[0], -1)
    # 1st fully connected layer + ReLU activation
```

```
    x = F.relu(self.fc1(x))
    # 2nd fully connected layer
    x = self.fc2(x)
    return x
```

Task 2 Class

```
# network class for task 2
# updated from homework guidelines
class HW4Net2(nn.Module):
  def __init__(self):
    super(HW4Net2, self).__init__()
    # 1st convolution layer
    self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)
    # pooling layer
    self.pool = nn.MaxPool2d(2, 2)
    # 2nd convolution layer
    self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
    # 1st fully connected layer
    self.fc1 = nn.Linear(8192, 64)
    # 2nd fully connected layer
    self.fc2 = nn.Linear(64, 5)

  def forward(self, x):
    # 1st convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv1(x)))
    # 2nd convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(x.shape[0], -1)
    # 1st fully connected layer + ReLU activation
    x = F.relu(self.fc1(x))
    # 2nd fully connected layer
    x = self.fc2(x)
    return x
```

Task 3 Class

```
# network class for task 3
# updated from homework guidelines
class HW4Net3(nn.Module):
  def __init__(self):
    super(HW4Net3, self).__init__()
    # 1st convolution layer
    self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)
    # pooling layer
```

```
    self.pool = nn.MaxPool2d(2, 2)
    # 2nd convolution layer
    self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
    # 10 extra convolution layers
    self.conv_extra = nn.ModuleList()
    for i in range(10):
      self.conv_extra.append(nn.Conv2d(32, 32, 3, padding = 1))
    # 1st fully connected layer
    self.fc1 = nn.Linear(8192, 64)
    # 2nd fully connected layer
    self.fc2 = nn.Linear(64, 5)

  def forward(self, x):
    # 1st convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv1(x)))
    # 2nd convolution layer + ReLU activation + pooling
    x = self.pool(F.relu(self.conv2(x)))
    # 10 extra convolution layers + ReLU activation
    for i in range(10):
      x = F.relu(self.conv_extra[i](x))
    x = x.view(x.shape[0], -1)
    # 1st fully connected layer + ReLU activation
    x = F.relu(self.fc1(x))
    # 2nd fully connected layer
    x = self.fc2(x)
    return x
```

Train Networks

```
# instantiate the three networks
Net1 = HW4Net1()
Net2 = HW4Net2()
Net3 = HW4Net3()

print("Number of Parameters:\nNet1: " + str(get_num_params(Net1)) + "\nNet2: " +
str(get_num_params(Net2)) + "\nNet3: " + str(get_num_params(Net3)))

# train the three networks
net1_training_loss = train_network(Net1)
net2_training_loss = train_network(Net2)
net3_training_loss = train_network(Net3)

# plot all training losses vs iterations
plt.figure()
plt.plot(net1_training_loss, label="Net1")
```

```
plt.plot(net2_training_loss, label="Net2")
plt.plot(net3_training_loss, label="Net3")
# set legend, figure title, and axes labels
plt.legend(loc="upper right")
plt.xlabel("Iterations")
plt.ylabel("Training Loss")
plt.title("Training Loss vs Iterations (10 Epochs)")
# show figure
plt.show()
```

Test Networks

```
# test each of the three networks
t_net1, p_net1 = test_network(Net1)
t_net2, p_net2 = test_network(Net2)
t_net3, p_net3 = test_network(Net3)

# calculate validation accuracy of each of the three networks
val_acc_1 = accuracy(t_net1, p_net1)
val_acc_2 = accuracy(t_net2, p_net2)
val_acc_3 = accuracy(t_net3, p_net3)
print("Validation Accuracy:\nNet1: " + str(val_acc_1) + "\nNet2: " +
str(val_acc_2) + "\nNet3: " + str(val_acc_3))

# calculate confusion matrices of each of the three networks
my_categories = ['boat', 'cake', 'couch', 'dog', 'motorcycle']
conf_mat_1 = cfd.from_predictions(t_net1, p_net1, display_labels =
np.array(my_categories), cmap='GnBu')
conf_mat_2 = cfd.from_predictions(t_net2, p_net2, display_labels =
np.array(my_categories), cmap='GnBu')
conf_mat_3 = cfd.from_predictions(t_net3, p_net3, display_labels =
np.array(my_categories), cmap='GnBu')
```